2
Sistemas Utilizados

O principal objetivo do projeto LuaPS é o desenvolvimento e a análise de um sistema publish-subscribe baseado em uma tabela hash distribuída. Para que pudéssemos focar as atenções nas decisões de implementação do sistema publish-subscribe e analisar as conseqüências da arquitetura e da infra-estrutura escolhida, utilizamos sistemas já existentes nas camadas inferiores do projeto. Antes de partirmos para discussões mais específicas sobre o sistema, iremos dedicar este capítulo à apresentação dos sistemas utilizados e à discussão das alterações e adaptações que se fizeram necessárias. Desta forma, teremos o embasamento necessário para analisar as decisões de projeto que foram tomadas no desenvolvimento do sistema publish-subscribe.

A primeira seção do capítulo descreve a linguagem de programação Lua e o sistema ALua, utilizado para prover a orientação a eventos. Nesta seção iremos, ainda, definir as alterações feitas no ALua para melhor adequá-lo ao sistema LuaPS. A segunda seção descreve o LuaPastry, que disponibiliza as funcionalidades necessárias para a criação da tabela hash distribuída.

2.1
Lua e ALua - Orientação a Eventos

Lua (Ierusalimschy et al., 1996) é uma linguagem de programação interpretada projetada com o objetivo de estender aplicações, embora também seja utilizada como linguagem de propósito geral. Lua utiliza programação procedural e tabelas associativas para representação de dados. Algumas outras características se destacam, como a tipagem dinâmica e o gerenciamento automático de memória com coleta de lixo.

O ALua (Ururahy et al., 2002) é uma extensão da linguagem Lua cujo objetivo é combinar o paradigma de orientação a eventos com uma linguagem de programação interpretada. A implementação do ALua define um mecanismo de comunicação entre processos baseado em eventos. Cada processo executa

um loop de código defino pelo ALua que verifica os sockets das conexões estabelecidas com outros processos. A comunicação se dá através da troca assíncrona de mensagens, onde cada mensagem é um trecho de código Lua. Quando o loop ALua detecta atividade em um socket, a mensagem é recebida e executada de forma atômica.

Um daemon deve ser executado em cada máquina, servindo como intermediário na troca de mensagens entre os processos, conforme esquematizado na figura 2.1 (Ururahy et al., 2002). Ao daemon cabem, dentre outra funções, as funções de criar e finalizar os processos, prover um espaço de nomes onde cada processo tenha um nome único e permitir a troca de mensagens.

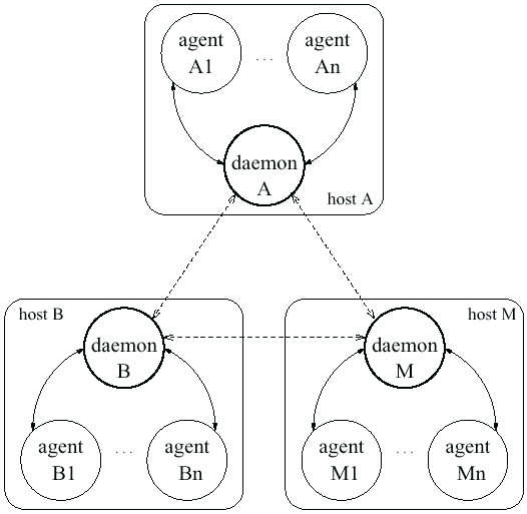

Figura 2.1: Modelo de Comunicação

Uma aplicação ALua é composta por um grupo de processos sendo executados em múltiplas máquinas e se comunicando utilizando a biblioteca LuaSocket. Cada processo possui um interpretador de comandos Lua e um loop de eventos que gerencia eventos de rede e os eventos da interface com o usuário.

Segue uma lista com as funções ALua e uma breve descrição:

- open(daemon) - Cria um novo daemon e se conecta a ele.
- close(daemon) - Fecha a conexão com um daemon.
- connect(daemon) - Se conecta a um daemon existente.
- link(app,daemons,callback) - Conecta daemons em uma aplicação.
- start(app,callback) - Inicia uma aplicação ALua.
- spawn(app,processes,callback) - Inicia processos na aplicação ALua.

– join(app,callback) - Entra em uma aplicação ALua.
– leave(app,callback) - Sai de uma aplicação ALua.
– query(app,callback) - Obtém informações sobre uma aplicação ALua.
– exit(proc,code,callback) - Termina processos ALua.
– loop() - Inicia o loop de eventos do ALua.

Uma extensão do ALua é o LuaTimer, que permite a inclusão de temporizadores em aplicações ALua. Para a manipulação de um temporizador, o LuaTimer oferece uma interface que permite associar strings de código Lua a freqüências de tempo. O código Lua fornecido será executado em intervalos regulares até que o temporizador seja removido.

Segue uma lista com as funções do LuaTimer e uma breve descrição:

– timeradd(callback,timer) - Associa a função ao temporizador.
– timerdel(timer) - Remove o temporizador.

2.1.1 Controle no Envio de Mensagens

A função *alua.send*, responsável pelo envio de mensagens, pode receber até três parâmetros no momento da execução. Além do endereço de destino e da mensagem a ser enviada, a função pode receber uma função de callback a ser executada no final do envio. Quando a função de callback é executada, esta recebe como parâmetro o estado final da transmissão, possibilitando que camadas superiores do sistema executem tratamentos de erro.

Quando uma mensagem é entregue com sucesso ou quando o ALua determina que houve um erro no envio da mensagem, a função de callback é executada e recebe como parâmetro o status da entrega da mensagem. Em função da implementação da biblioteca de sockets utilizada, entretanto, existem situações onde o insucesso na entrega não é determinado. Nestes casos, a função de callback é executada com o status da entrega da mensagem inconsistente, impossibilitando um correto controle de erros. No sistema LuaPS desenvolvido, o controle de erros no envio das mensagens precisava ser executado corretamente, principalmente ao considerarmos a implementação de

conexões persistentes. Desta forma, precisamos realizar algumas alterações no ALua para garantir a execução correta do controle de erros.

A primeira alteração feita foi a inclusão de um novo parâmetro na função de envio de mensagens. Este parâmetro é um número que indica quanto tempo o ALua irá esperar até receber a confirmação de entrega de uma mensagem. Caso a confirmação não seja recebida neste intervalo de tempo, o ALua executará a função de callback indicando a situação ocorrida. Para que o controle de tempo seja realizado, o envio das mensagens precisou ser alterado, passando a confirmar as mensagens recebidas.

A possibilidade de execução de funções no ALua sem a passagem de todos os parâmetros indicados pela função é bastante interessante neste caso. A alteração realizada considera, caso o parâmetro de tempo não seja passado, que o controle de timeout não será realizado. Desta forma, a alteração no ALua é transparente para sistemas anteriormente desenvolvidos, que continuarão a funcionar normalmente.

2.1.2 Console para Interface com o Usuário

Uma aplicação publish-subscribe deve possibilitar interação com os usuários a qualquer momento, permitindo a execução das funcionalidades conforme suas necessidades. O ALua, entretanto, não disponibiliza uma interface para interação com o usuário, se baseando apenas na detecção de eventos de socket. Para permitir a interação do usuário, natural para diversas aplicações, desenvolvemos um console que adapta o modelo de eventos do ALua. Cabe ao console receber os comandos do usuário e transformá-los em eventos reconhecidos pelo ALua.

Qualquer processo previamente iniciado pode instanciar um console de interação. Para isto, o módulo do console deve ser carregado e o método de inicialização deve ser executado. O método de inicialização do console irá iniciar um novo processo que estará conectado ao processo original. Este novo processo, ao contrário dos processos ALua normais, não executará o loop do ALua, e sim um loop particular. Ao contrário do loop ALua, que determina eventos de socket, o loop particular irá verificar eventos do teclado. Desta forma, o console não estará apto a receber mensagens via socket, sendo possível apenas o envio das mensagens.

A figura 2.2 ilustra a arquitetura da aplicação com a utilização do console. A aplicação ALua do exemplo instanciou dois processos. O processo A

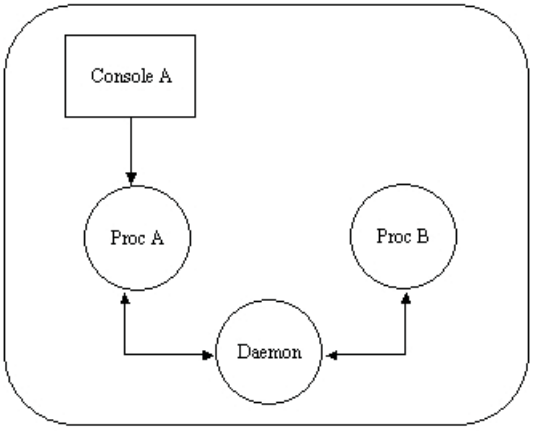

Figura 2.2: Console de Interação com o Usuário

instanciou um console enquanto que o processo B não o fez. Podemos notar na figura que a comunicação entre o console e o processo se dá apenas na direção especificada.

Uma vez que a necessidade de interação é natural para muitas outras aplicações, procuramos dividir a implementação do console em uma parte genérica, compatível com qualquer aplicação, e uma parte específica, que deve ser definida para cada aplicação desenvolvida. Para especificar a parte particular, iremos utilizar funções passadas como parâmetros na instanciação do console.

Na instanciação de um novo console, pode-se passar como parâmetro três funções particulares que irão definir o comportamento do console. A primeira função passada como parâmetro será executada antes do loop do console e serve para executar algum tratamento que deva ser feito antes da inicialização do loop. A segunda função passada como parâmetro será executada sempre que um comando do usuário for detectado e tem a função de traduzir o comando do usuário em um evento de socket que será capturado pela aplicação ALua. A terceira função passada como parâmetro será executada quando o loop do console for terminado e serve para executar algum tratamento de finalização que possa ser necessário.

O loop do console propriamente dito tem a funcionalidade básica de detectar eventos do teclado, identificar os comandos desejados pelo usuário e traduzir os comandos para o ALua como eventos de socket. Se um comando de saída for executado pelo usuário - atualmente definido como o comando *exit* - o loop será encerrado e o processo terminado. Qualquer outro comando identificado é enviado para o processo através do socket que conecta o processo e o console.

Segundo a arquitetura ALua definida, cada socket aberto e verificado pelo loop do ALua possui uma função de tratamento a ele associada. A função de tratamento associada ao socket existente entre o console e o processo combina uma parte genérica com a função particular passada como segundo parâmetro. Desta forma, para cada comando recebido pelo processo, a função particular é executada com o comando passado como parâmetro, permitindo um tratamento particular para cada aplicação.

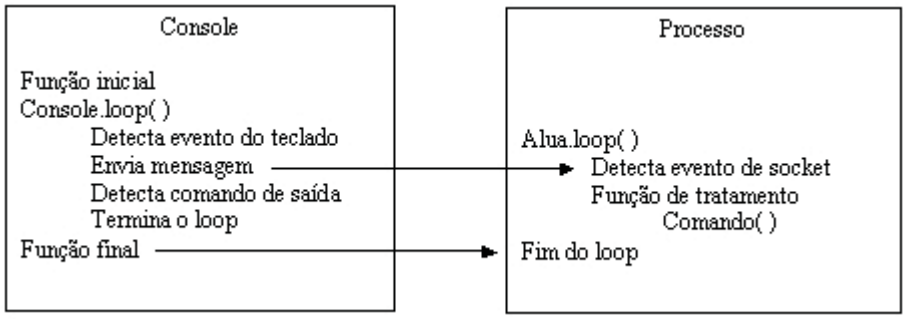

Figura 2.3: Comunicação Console-Processo

A figura 2.3 ilustra o processo descrito. Na figura, o console executa o loop particular e o processo executa o loop do ALua. Quando o console identifica um comando, o mesmo é enviado para o processo. O processo identifica o evento de socket e executa a função particular passando o comando como parâmetro. Desta forma, o comando pode ser executado com sucesso. Quando o console identifica o comando de fim de loop, o loop do console é encerrado e os processos terminados.

Embora tenhamos implementado o console considerando um loop independente e sem a possibilidade do recebimento de mensagens, existem outras soluções que poderiam ser implementadas e que, para outras situações, poderiam ser mais interessantes.

A primeira sugestão que iremos abordar está relacionada à necessidade de recebimento de retorno originado nos processos por parte do console. Na implementação disponibilizada, o loop individual do console não considera a existência de sockets para o recebimento de mensagens. Uma alteração simples na implementação, visando permitir o retorno na comunicação, seria a inicialização de sockets para comunicação neste sentido e a alteração no loop do console, incluindo a verificação de atividade neste sockets. Desta forma, a comunicação entre o console e os processos poderia acontecer nos dois sentidos.

Uma segunda possibilidade, um pouco diferente da solução implementada, considera a utilização do loop ALua já existente. Para que o loop do

ALua possa funcionar corretamente e verificar a atividade nos sockets, não podemos iniciar nenhum processamento que não termine. Ou seja, a verificação constante e tratamento dos eventos de teclado deve ser interrompido de tempos em tempos.

Uma vez que a verificação de eventos do teclado é bloqueante, o único momento onde a verificação pode ser interrompida é após a detecção da disponibilidade de um comando. Neste momento, o comando deve ser tratado e a verificação deve ser interrompida permitindo que o loop ALua verifique atividade nos sockets. Vamos considerar, portanto, uma função que busca por comandos do teclado e realiza os respectivos tratamentos. Em seguida a função envia uma mensagem para o próprio console solicitando a execução da própria função. Neste momento, a função termina e o loop do ALua pode verificar atividade nos sockets. O loop do ALua irá, então, tratar o recebimento de mensagens que tenham sido recebidas, incluindo a mensagem que reinicia o processo de verificação do teclado. Desta forma, usamos o loop do ALua e criamos uma função que simula o loop do console.

2.2 Lua Pastry - Tabela Hash Distribuída (DHT)

Sistemas publish-subscribe podem ser implementados de forma centralizada ou distribuída. Podemos encontrar uma discussão a respeito de vantagens e desvantagens de cada uma das abordagens em (Tam et al., 2003). Sistemas publish-subscribe centralizados têm como principal vantagem a possibilidade de acesso a uma imagem global das informações, permitindo a utilização de algoritmos avançados para a obtenção de informações. Podemos citar, por exemplo, o sistema S-ToPSS (Petrovic et al., 2003), que desenvolve uma metodologia para executar filtragem semântica nas notificações. Por outro lado, sistemas centralizados apresentam desvantagens quando consideramos escalabilidade ou tolerância a falhas.

Aplicações peer-to-peer têm se popularizado bastante, principalmente com a utilização de sistemas de troca de arquivos amplamente difundidos na Internet. Estudos sobre aplicações peer-to-peer têm oferecido arquiteturas escaláveis, confiáveis e tolerante a falhas, podendo servir de base para o desenvolvimento de sistemas distribuídos descentralizados. As tabelas hash distribuídas se destacam como arquiteturas deste tipo, oferecendo uma plataforma atrativa para o desenvolvimento de sistemas publish-subscribe. Podemos encontrar uma análise mais detalhada das carac-

terísticas de DHTs em (Rowstron & Druschel, 2001), (Ratnasamy et al., 2001) e (Stoica et al., 2001), onde os conceitos de tabelas hash distribuídas são apresentados.

Tabelas hash distribuídas já foram utilizadas com sucesso em diversas áreas, destacando-se, por exemplo, o desenvolvimento de sistemas de arquivos distribuídos, como em (Dabek et al., 2001) e (Muthitacharoen et al., 2002). Ao considerarmos sistemas publish-subscribe, arquiteturas deste tipo também oferecem uma plataforma bastante atrativa, principalmente se considerarmos sistemas baseados em tópicos, como o Scribe (Rowstron et al., 2001).

Sistemas baseados em conteúdo não se adequam à utilização de tabelas hash distribuídas tão naturalmente. Apesar disto, existem estudos sobre o assunto, como em (Terpstra et al., 2003). A principal idéia utilizada em sistemas como este é a tradução de filtros por conteúdo para filtros por tópicos. Para isto, os sistemas definem uma estrutura básica de campos que deve ser utilizada na definição do filtro, limitando a liberdade de sua definição. Após a definição do filtro, o sistema cria tópicos utilizando os valores definidos pelo usuário. Desta forma, embora a aplicação seja baseada em conteúdo, a infra-estrutura considera a utilização de tópicos.

O Pastry (Rowstron & Druschel, 2001) é um protocolo que foi desenvolvido com o objetivo ser um substrato para a construção de aplicações peer-to-peer. O protocolo, baseado em uma tabela hash distribuída, provê um esquema de localização e roteamento baseado em uma rede auto-organizável de nós. O Pastry implementa uma arquitetura completamente descentralizada, resistente a falhas, escalável e confiável.

No protocolo Pastry, um ID de 128 bits é atribuído a cada nó da rede e cada mensagem tem uma chave de mesmo tamanho. Para efeitos de roteamento, os IDs dos nós e chaves das mensagens são interpretados como uma seqüência de dígitos na base 2^b, onde b é um parâmetro a ser escolhido pelo desenvolvedor. Na implementação do sistema LuaPastry, o parâmetro b foi definido como tendo o valor 4, de modo que os IDs e as chaves serão uma seqüência de dígitos na base 16.

As mensagens são encaminhadas para o nó cujo ID seja numericamente mais próximo à chave de endereço das mensagens. Para que as decisões de encaminhamento possam ser tomadas, os nós armazenam uma tabela de roteamento, um conjunto de vizinhança e um conjunto de folhas.

Considerando uma rede com N nós, a tabela de roteamento de um nó é organizada em $log_{2^b} N$ linhas com $2^b - 1$ entradas em cada linha. As entradas em uma linha n irão referenciar nós cujos IDs terão n dígitos em comum com o nó em questão, mas cujo valor do dígito n+1 será um dos $2^b - 1$ valores diferentes

do dígito n+1 do nó em questão. Dentre os possíveis nós pertinentes para uma entrada, o nó escolhido será o nó que mais se adequar segundo alguma métrica escolhida. Caso nenhum nó seja pertinente a uma entrada, a entrada é deixada vazia.

O conjunto de folhas terá L entradas, onde o valor L, outro parâmetro a ser escolhido pelo desenvolvedor, terá o valor 2 nesta implementação. Neste conjunto, serão armazenados os $L/2$ nós com os maiores e menores IDs mais próximos do ID do nó em questão.

O conjunto de vizinhança terá M entradas, referenciando os vizinhos mais próximos segundo a métrica escolhida. O parâmetro M foi definido como tendo o valor 4. O conjunto de vizinhança não é utilizado no roteamento e será explicado mais adiante.

Em cada passo do roteamento, o nó que deve rotear a mensagem determina o prefixo em comum entre seu ID e o endereço da mensagem. Em seguida, é feita uma pesquisa na sua tabela de roteamento em busca de um nó cujo prefixo em comum com o endereço da mensagem tenha um dígito a mais que o prefixo atual. Se este nó não for localizado, o conjunto de folhas é verificado em busca de um nó cujo prefixo seja tão longo quanto o atual, mas que seja numericamente mais próximo à chave. Se este nó também não puder ser localizado, a mensagem já terá atingido o nó de destino.

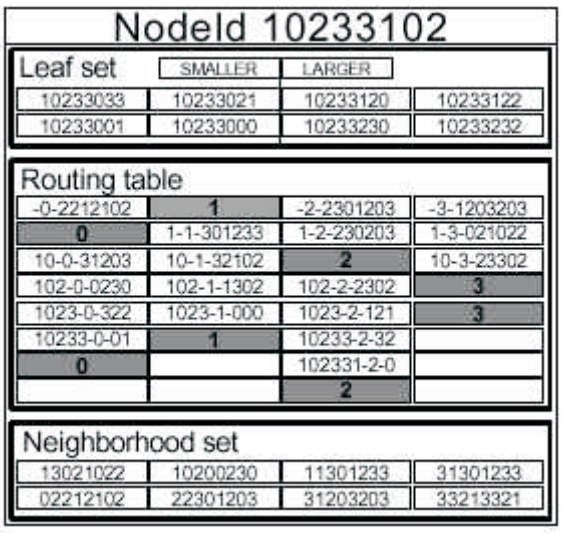

Figura 2.4: Tabela de Roteamento para o Nó com ID 10233102

A figura 2.4 (Rowstron & Druschel, 2001) representa a tabela de roteamento para um nó com ID 10233102. No exemplo, $b = 2$ e $L = 8$. A linha superior da tabela de roteamento referencia nós cujos IDs não têm nenhum dígito em comum com o ID do nó em questão. Desta forma, as entradas desta

linha não possuem nenhum dígito antes do primeiro hífen. Entre o primeiro e o segundo hífen, podemos ver todos os possíveis valores que esta posição pode receber. Uma vez que o ID do nó possui o valor 1 nesta posição, esta entrada não é preenchida na tabela. Após o segundo hífen, temos o restante dos dígitos dos nós que são diferentes dos dígitos do ID do nó em questão. As linhas seguintes possuem a mesma estrutura, terminando na última linha, que referencia nós cujos IDs têm sete dígitos em comum com o ID do nó em questão.

Ao analisarmos o processo de roteamento de uma mensagem, podemos notar que cada mensagem será entregue em, no máximo, $log_{2^b} N$ passos. Isto porque, a cada passo, o prefixo em comum entre a chave da mensagem e código do nó terá um dígito a mais. Sendo assim, a escolha do parâmetro b estabelece o relacionamento entre o número de entradas povoadas da tabela de roteamento de um nó $(log_{2^b} N)(2^b - 1)$ e o número máximo de saltos no roteamento $log_{2^b} N$.

Quando um novo nó deseja ingressar na rede Pastry, ele precisa inicializar sua tabela de roteamento de forma consistente e avisar aos outros nós da rede a sua existência. Para isto, iremos assumir que o novo nó conhece um nó que já pertença a rede e utiliza o ID deste nó na função de entrada na rede.

Vamos considerar que um novo no X, que conhece o nó A da rede, deseja ingressar na rede Pastry. Para isto, o nó X solicita ao nó A que roteie uma mensagem de *join* na rede. A chave desta mensagem será o próprio valor X. Assim como todas as mensagens roteadas pelo Pastry, a mensagem de *join* irá atingir o nó Z que possui ID mais próximo de X. Como resposta à mensagem, o nó Z, o nó A e todos os nós do caminho entre A e Z enviam suas tabelas de roteamento para o novo nó X. A partir das tabelas recebidas, o novo nó X será capaz de inicializar suas tabelas utilizando o protocolo definido em (Rowstron & Druschel, 2001). Quando a tabela de X estiver pronta, o nó envia uma mensagem para todos os nós em sua tabela de roteamento, conjunto de folhas e vizinhança. Com estas informações, os nós atualizam suas tabelas levando em consideração a existência do novo nó X.

Quando um nó X identifica um nó A do conjunto de folhas que tenha falhado, X inicia o processo de substituição da entrada. Inicialmente, X identifica se a entrada A está presente no subconjunto de folhas com IDs maiores ou menores do que X. Identificado o subconjunto de folhas adequado, X solicita ao nó de maior ID no dado subconjunto a sua tabela de folhas. Com os valores recebidos, X identifica uma nova entrada, testa a atividade do novo nó e substitui a entrada em sua tabela. Este processo garante que entradas possam ser substituídas a não ser que $L/2$ entradas adjacentes falhem simultaneamente.

Quando um nó X precisa substituir uma entrada R_l^d da tabela de roteamento, X solicita a entrada a um outro nó em uma posição $R_l^i, i \neq d$ da tabela de roteamento. Caso nenhum nó da linha l seja capaz de retornar uma entrada válida para substituir o nó em falha, X prossegue o procedimento solicitando a entrada para os nós na linha *l+1*.

O conjunto de vizinhança, não utilizado no processo de roteamento, é importante no processo de manutenção dos nós ativos. Periodicamente, os nós verificam a atividade de seus vizinhos e os substituem no caso de falhas.

O LuaPastry (Teixeira & Rodriguez, 2005) é uma implementação do protocolo Pastry para o ambiente ALua. O objetivo do projeto LuaPastry é disponibilizar ao usuário uma infra-estrutura de rede baseada em Tabela Hash Distribuída para implementação de aplicações Peer-to-Peer. O programa fornece uma camada de abstração para o usuário, facilitando a implementação de aplicações distribuídas que virão a se beneficiar das características do protocolo Pastry. O sistema implementado disponibiliza funções para a entrada em uma rede existente ou a criação de uma nova rede e para o encaminhamento de uma mensagem.

Segue uma lista com as funções disponibilizadas pelo LuaPastry e uma breve descrição:

- JoinNetwork(no) - Inicia ou entra em uma rede Pastry.
- JoinComplete() - Função executada ao fim da entrada do nó na rede.
- RouteMsg(hash,msg) - Roteia mensagem para o nó gerente da chave.
- GetMyNodeId() - Retorna o id do próprio nó.
- MyState() - Retorna tabela com informações sobre o estado do nó.